Cache organization on loop blocking

François Bodin, André Seznec

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Cache Organization Influence on Loop Blocking

François Bodin, André Seznec

N° 2255

Mars 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

Cache Organization Influence on Loop Blocking

François Bodin, André Seznec*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Calcpar

Rapport de recherche n° 2255 — Mars 1994 — 21 pages

Abstract:

Performance tuning on today's computers has become very complex. One of the factor of this complexity is the use of memory hierarchies, and particularly of cache memories. Code transformations such as loop blocking are used for improving temporal locality in numerical codes. Unfortunately, the behavior of direct-mapped caches and set-associative caches are very sensitive to parameters such as the respective placement of arrays determined by the leading sizes of arrays. This leads sometimes to unpredictable and catastrophic performance even on blocked numerical kernels. Most users are not expert in cache organizations and cannot be aware of such phenomena.

In this paper, we show that the recently proposed 4-way skewed associative cache is quite insensitive to array placements in memory, and then provides to the user a quite stable and predictable behavior on the basic algorithms as well as blocked algorithms. The average behavior of the 4-way skewed-associative cache is also better than the average behavior of the 4-way set-associative cache on all algorithm versions.

When using the 4-way skewed associative cache, copying is never necessary for getting predictable performance while it is generally the only mean to get such predictable performance on set-associative and direct-mapped caches.

Furthermore, on blocked algorithms, a large fraction of the cache space in a 4-way skewed associative cache may be used for blocking the loop, thus leading to a limited overhead due to blocking.

Key-words: cache, predictable performance, loop blocking, skewed-associative caches

(Résumé : tsvp)

*e-mail :bodin, seznec@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

De l'influence de l'organisation des caches sur le blocage des boucles

Résumé : Les performances des ordinateurs d'aujourd'hui sont devenues très difficiles à exploiter et à prévoir. Un des facteurs rendant cette prévision extrèmement complexe est l'utilisation de hiérarchies mémoires, et en particulier de mémoire caches. Des transformations de programmes telles que le blocage de boucles peuvent être utilisées pour améliorer la localité des applications dans les codes numériques. Malheureusement, le comportement des caches à correspondance directes et des caches associatifs par ensemble sont très sensibles à des paramètres tels que le placement des tableaux en mémoire ; ceci entraine parfois des chutes de performances imprédictibles et catastrophiques même sur des codes bloqués. La plupart des utilisateurs ne peuvent pas être conscients de tels phénomènes.

Dans cet article, nous montrons que le cache associatif brouillé 4 voies que nous avons récemment proposé est à peu près insensible au placement relastif des tableaux en mémoire et qu'il fournit ainsi une performance prédictible et stable à l'usager.

De plus, sur les algorithmes bloqués, une partie importante de l'espace d'un cache associatif brouillé peut être utilisé pour les données réutilisables ; ceci limite le surcoût lié au blocage de boucles.

Mots-clé : cache, performance prévisible, blocage de boucles, caches associatifs brouillés

1 Introduction

Modern processors reference data through caches; as the penalty on misses becomes higher and higher, performance of these processors dramatically depends on the cache miss ratios.

Limiting cache miss ratio is a major issue for achieving high performance. Misses are generally classified in three categories [7]: *first reference misses*, *capacity misses* and *conflict misses*. Capacity misses are due to the limited size of the cache: all the data which will be reused in the future cannot be kept at the same time in the cache. *Conflict misses* are due to the lack of associativity of the cache (too many data blocks conflicting for a single set) and the non-optimality of the cache replacement policy [17].

Reducing capacity misses and conflict misses in numerical applications by software technique has been addressed in many studies. First studies [5, 20, 21, 11, 4] focused on limiting the size of the current working set of the applications, thus reducing the number of capacity misses on the cache. Blocking and unimodular transformations s can be used for enhancing spatial and temporal locality in applications.

Unfortunately, as caches are not perfect, loop transformations applied to reduce the size of current working set in such a way that it becomes smaller than the cache size are not sufficient; performance may suffer in a quite unpredictable way from conflict misses [10]. For instance, in a recent study, Schlansker et al [12] showed that, even on very regular memory access patterns such as iterating on the read of a fixed size memory subblock, the miss ratio on a 32-way set-associative cache dramatically depends on parameters such as the number of rows of the whole matrix in a quite unpredictable way : in their example, depending on whether the number of rows is 2727 or 2729, nearly all the accesses result in a hit or nearly all the access results in a miss.

For most users, such unpredictable behaviors can not be accepted. Then predictable and stable performance is a major issue. Blocking is not a sufficient technique for many applications and many cache configurations. In order to overcome this difficulty, blocks exhibiting high level of reuse may be copied in order to control the respective placement of data in memory and then avoid placement conflicts in the cache [19, 5, 10]; unfortunately copying is not always possible and may induce large overhead on many numerical kernels.

In order to avoid unpredictable and catastrophic behavior of caches without copying, Schlansker et al [12] proposed to randomize the set selection from address in order to obtain a predictable and good behavior. Nevertheless their proposal suffers from two major drawbacks; first a high degree (in the 16-32 range) of associativity is needed and at a second point, some complex hardware mechanism is needed to pseudo-randomize the set selection in the cache.

Recently, the skewed-associative cache, a new associative cache structure has been proposed in [13, 14]. In this paper, we investigate the sensitivity of the behavior of the skewed-associative cache to parameters such as size of arrays in numerical kernels on dense structures. Unlike usual set-associative caches, the behavior of a 4-way skewed-associative cache is quite insensitive to those parameters of the applications. This leads to better average miss ratio than set-associative cache and more predictable performance.

When using direct-mapped caches or set-associative caches, copying is generally the only viable software solution to avoid unpredictable and catastrophic behaviors for some dimensions of the arrays; when using a 4-way skewed associative cache, blocking is sufficient, thus extra computation cost for copying the arrays is avoided.

Moreover, our simulations also establish that, even when restructuring technique such as blocking and copying are used, performance of direct-mapped caches are significantly worse than performance of set and skewed associative caches.

On usual direct-mapped or set-associative caches, the behavior of caches on blocked algorithms degrades very rapidly when the blocking factor increases, experiments have also been conducted which indicates that a larger fraction of the cache size may be considered for determining the blocking factor for a 4-way skewed-associative cache than for a usual set-associative cache organization.

The remainder of the paper is organized as follows. In Section 2, we recall the principles of a skewed-associative cache; in Section 3 we present a very simple experiment which explains why skewed-associative cache should exhibit a better average behavior than a usual set-associative cache. In Section 4, we present the simulation tools which have been used in the paper. In Section 5, the impact of varying sizes of arrays is studied on a few numerical kernels; original, blocked and blocked copied algorithms are studied. In Section 6, we try to characterize the fraction of the cache size that is available for blocking on set-associative caches as well as on skewed-associative caches. Section 7 summarizes this study.

2 Skewed-associative caches

2.1 Principle

Skewed associative caches have been recently proposed in [13, 14]. A set-associative cache is illustrated by Figure 1: a X way set-associative cache is built with X distinct banks. The memory block at address D may be physically mapped on physical line $f(D)$ in any of the distinct banks. This vision of a set-associative cache fits with its physical implementation: X banks of static memory RAMs.

In a skewed associative cache (Figure 2), different mapping functions are used for the distinct cache banks i.e., a memory block at address D may be mapped on physical line $f_0(D)$ in cache bank 0 or in physical line $f_1(D)$ in cache bank 1, etc.

It has been shown in [13, 14] that, on general applications, skewed-associative caches exhibit an average lower miss ratio than set-associative caches.

2.2 Choosing skewing functions

When designing a skewed associative cache, the mapping functions may be chosen in order to minimize conflict misses and hardware cost. We list some of these properties [13, 14].

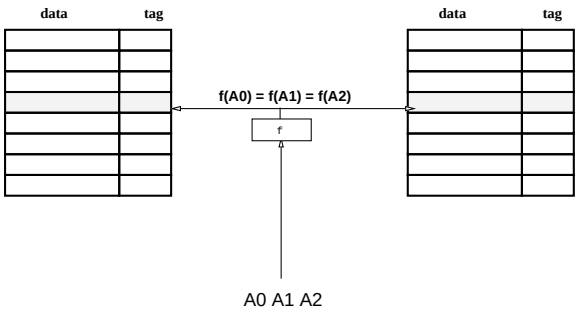


Figure 1: 3 data blocks conflicting for a single set on a two-way set-associative cache

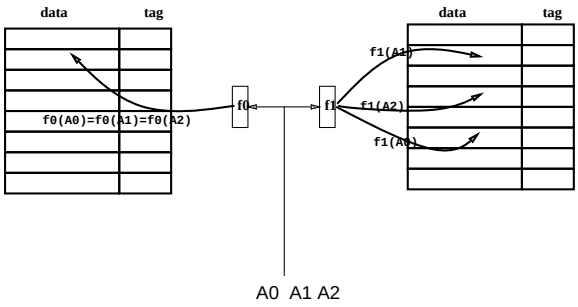


Figure 2: Data blocks conflicting for a cache line on bank 0, but not on bank 1 on a skewed-associative cache

Inter-bank dispersion In a usual X-way set-associative cache, when (X+1) data blocks contend for the same set in the cache, they are all conflicting for the same line in the X cache banks: one of the blocks must be rejected from the cache (Figure 1).

Skewed-associative caches avoid such a situation by scattering the data: mapping functions can be chosen such that whenever two data blocks conflict for a single location in cache bank i, they have very low probability to conflict for a location in cache bank j (Figure 2).

Local dispersion in a single bank Many applications exhibit spatial locality, therefore the mapping functions must be chosen so as to avoid having two "almost" neighbor data blocks of data conflicting for the same physical cache blocks in cache bank i.

The different mapping functions must respect a certain form of local dispersion on a single bank; the mapping functions f_i must limit the number of conflicts when mapping any region of consecutive data blocks in a single cache bank i.

Simple hardware implementation A key issue for the overall performance of a processor is the pipeline length. Using distinct mapping functions on the distinct cache banks will have no effects on the performance, as long as the computations of the mapping functions can be added to a non critical stage in the pipeline and do not lengthen the pipeline cycle.

2.3 An example of skewing functions

We present here the skewing functions which were used in the simulations illustrated in this paper. These skewing functions are based on XORing some bits in the address of a memory block (as in [13, 14]).

Let us consider a skewed associative cache built with 2 or 4 cache banks, each one consisting of 2^n cache lines of 2^c bytes,
let $\rho^{(n)}$ be the bit reversal on n bits, let Mask1=0x5555555 and Mask2=0xaaaaaaa,
data block at memory address $A_3 2^{c+2n} + A_2 2^{n+c} + A_1 2^c$ may be mapped:

1. on cache line $A_1 \oplus \rho^{(n)}(A_2)$ in cache bank 0
2. *or* on cache line $A_1 \oplus A_2$ in cache bank 1
3. *or*[1] on cache line $A_1 \oplus (\rho^{(n)}(A_2).Mask1 \oplus A_2.Mask2)$ in cache bank 2
4. *or* on cache line $A_1 \oplus (\rho^{(n)}(A_2).Mask2 \oplus A_2.Mask1)$ in cache bank 3

These functions satisfy the criterion for "good" skewing functions defined in [13] (inter-bank dispersion, local dispersion and low hardware costs).

A pseudo-LRU replacement policy similar as that described in [14] was used in the simulations of the skewed-associative cache.

[1] on a 4-way skewed associative cache

3 How a skewed-associative cache handles conflict misses

We have conducted a very simple experiment to illustrate the benefits that can be expected with a skewed-associative cache instead of a usual set-associative cache.

Let us consider a cache consisting of 512 cache lines. Let us consider a sequence of X data blocks with random addresses. This sequence is iteratively read 10 times. Five cache configurations were simulated: direct-mapped, 2 and 4-way set-associative, 2 and 4-way skewed-associative. The pseudo-LRU replacement policy was used for the skewed-associative cache[2]. The experiment was repeated on 100 different sequences for every sequence size.

3.1 Data dispersion

Figure 3a illustrates the average ratio of the blocks of the sequence that remain valid in the cache after a single read of the whole sequence for sequence sizes varying from 32 to 512.

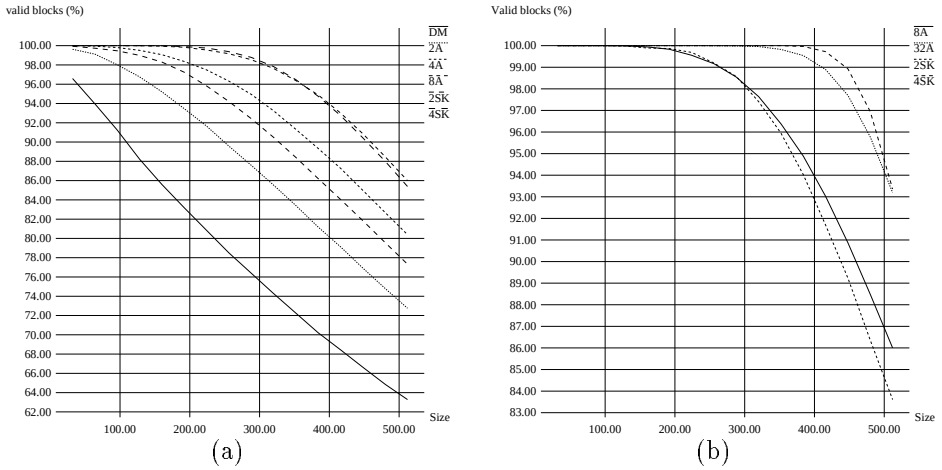


Figure 3: (a) Ratio of valid blocks after one read, (b) Ratio of valid blocks after ten reads

The number of valid blocks in the 2-way skewed-associative cache is larger than the number of valid blocks in the 2-way set-associative cache and slightly lower than the number of valid blocks in the 4-way set-associative cache.

The number of valid blocks in the 4-way skewed-associative cache is approximately equal to the number of blocks valid in the 8-way set-associative cache, but is lower than the number of valid blocks in 16-way and 32-way set-associative caches.

[2]For the set-associative cache, the parameter measured in this experiment does not depend in any way of the replacement policy

Then after a single read sequence, at equal associativity degrees, more data are valid on a skewed-associative cache than on a set-associative cache.

3.2 Self data reorganization

A second phenomenon accentuates the advantage of the skewed-associative over the set-associative cache.

Figure 3b illustrates the average ratio of the blocks in the sequence that remain valid in the cache after ten successive reads of the whole sequence for sequence sizes varying from 32 to 512. For direct-mapped and set-associative caches, the number of valid blocks does not vary after the first sequence of reads: when a block is missing, its target set was full and then another block must be invalidated before loading it.

In the skewed-associative cache, the number of data blocks present at the same time in the cache depends on the precise mapping of each data block in the cache. Among the other possible locations for a data block D present in the cache at time t, there may be an empty location; block D may be removed from the cache by a miss on an other block D'; in this case, the next time D will be referenced, D can be mapped in the empty location and thus the number of data alive at the same time in the cache can increase.

For instance, after ten iterations on reading the whole sequence, the number of valid blocks in the 4-way skewed-associative cache (resp. 2-way) is in the same range as the number of valid blocks in the 32-way set-associative (resp. 8-way).

In blocked algorithms, block sizes are chosen in such a way that the size of the reused data is smaller than the cache size. It may be expected that the self data reorganization in the skewed-associative cache will limit conflict misses on such blocked algorithms.

4 Evaluation methodology

In order to capture effective program behavior including loop management, scalar references, etc, we chose to use effective program execution traces.

The *Spa* package developed by Gordon Irlam [9] was used to generate address traces for programs executed on a SUN SparcStation10. F77 Fortran compiler with -O4 optimization was used.

No modification of the binary code to be analyzed was required; user code of a single application can be completely traced except for the OS kernel code. As we studied the behavior of numerical kernels consisting of a few nested loops, only data references were piped on a cache simulator.

Five cache organizations were simulated in a single path: direct-mapped, 2-way and 4-way set-associative, 2-way and 4-way skewed-associative caches.

In order to limit the sizes of the problems needed to exceed cache capacity (and then simulation time), a 8Kbyte cache was simulated. The cache line size chosen for simulation was 32 bytes.

Sensitivity to Parameter Variations In order to measure the sensitivity of the cache behavior to parameters such as array sizes or block sizes, systematic experiments were repeated with varying only block size and leading size (i.e. number of rows in a matrix).

5 Software technique and cache organizations

As previously mentioned, the impact of the software technique proposed for improving data locality on the behavior of caches is not well understood.

The experiments presented in this section evaluate the performance of cache for three loop kernel : 100×100 matrix-matrix multiply, 340×340 2D Jacobi loop and 100×100 LU factorization. In all the experiments, we vary the leading dimension of the arrays used in the loops to highlight the impact of the array declaration on the cache behavior.

For all the kernels we simulated several versions of the kernel (original, blocked and copy blocked) which were traced and piped through a cache simulator. We measured the impact on these software technique on the number of cache misses, on extra memory references and on extra instructions. For **blocked** and **copy blocked** versions of the applications, the block size was chosen in order that the total size of the blocks is approximately equal to half of the cache size; as shown in Section 6, this happens to be a good approximation.

5.1 Matrix-matrix multiply

A 100×100 matrix-matrix multiply was simulated. The three versions of the programs which were simulated are illustrated in Figure 4. The number of floating-point references in the different versions of the algorithm is predictable from the Fortran code:

- in the original loop, each element of matrix A is invariant in the inner most loop and then is read and written one time (20000 references) and each element of matrices B and C are read 100 times (2000000 references).
- when blocking, each element of matrix A is read and written five times instead of a single time in the original loop, thus leading to 80000 more memory accesses than in the original loop (see Figure 4).
- while in the copy blocked version, each element of matrix C is copied one time (20000 references) and each element of matrix B is copied five times (100000 references).

The number of instructions in the different versions highly depends on the quality of the F77 compiler. On our examples, the F77 compiler unrolls the inner most loop 4 times. On the blocked and copied version, it uses only 21 instructions for coding this unrolled loop; as it uses 23 instructions for coding the main loop in the original algorithm and the blocked algorithm, more instructions are executed for the blocked algorithm than for the blocked and copied algorithm. Statistics on the execution of the three simulated algorithms are reported in table 1.

	Original	Blocked	Bl & Co
floating point ref	2020000	2100000	2220000
data ref	2179677	2267224	2387747
instructions	6873637	8172743	8055858

Table 1: Characteristics on the different matrix-multiply versions

The data reuse in the 100×100 matrix-matrix multiply is very high: each element of matrix B and C are used 100 times, moreover each cache block contains 4 words, then leading to 400 reads of a single cache blocks. Such an optimal reuse can only be obtained when the whole matrices fit in the cache. For the blocked and blocked copied versions, a relatively correct estimation of the minimal number of misses is obtained by assuming a perfect cache but no reuse across the blocks: 27500 misses.

Figure 5 illustrates the number of misses for each of the three versions run with row numbers of the arrays varying from 100 to 512[3].

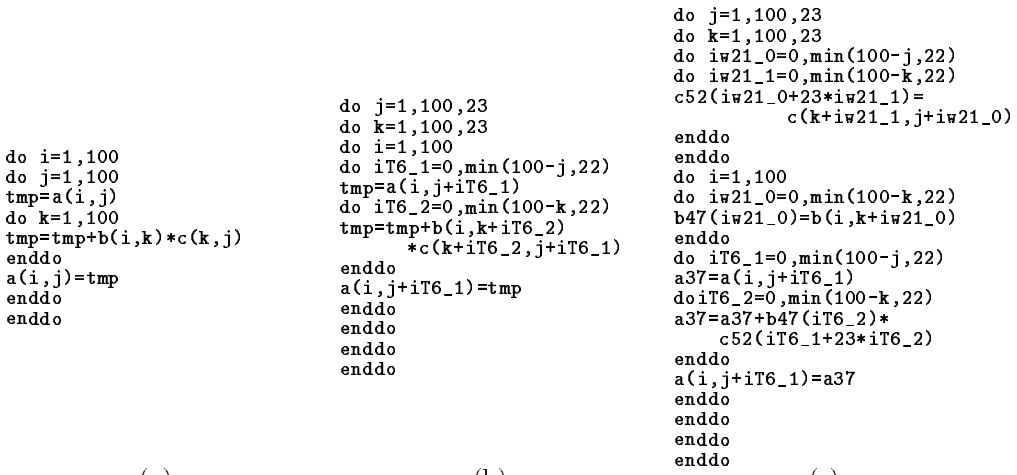
Original loop and blocked loop: It clearly appears that direct-mapped and 2 and 4-way set-associative caches exhibit quite unpredictable behaviors on the original version as well as on the blocked version of the algorithm. The number of misses on the 2-way skewed-associative cache is less irregular, and becomes quite regular on the 4-way skewed-associative cache. The average miss ratio is also better on a 4-way skewed-associative cache than on the other cache structures: for the blocked version, the average miss number is around 60000 for the 4-way skewed-associative cache against 130000 for the 4-way set-associative cache.

Blocked and Copied: Associating blocking and copying allows to obtain relatively stable numbers of misses on the matrix multiply. Nevertheless it is noticeable that the 4-way skewed-associative cache outperforms the other cache organizations, and particularly the average number of misses on the direct mapped cache is about three times higher than the average number of misses on the 4-way skewed-associative cache.

5.2 2D Jacobi Loop Kernel

The kernel studied here is a 2D Jacobi loop extracted from an application, called PENAL [2], that computes permeability in porous media using a finite difference method. The original loop nest is shown in Figure 6. Due to lack of space , blocked and blocked copied versions of the kernel are not illustrated. In this kernel, the data reuse is more limited than in the

[3]On all experiments, the whole program is traced including entry in the program, table initializations, .. , this add some extra misses; for instance, an empty program generates about 55000 data references and around 10000 data misses



```
do i=1,100
do j=1,100
tmp=a(i,j)
do k=1,100
tmp=tmp+b(i,k)*c(k,j)
enddo
a(i,j)=tmp
enddo
enddo
```

(a)

```
do j=1,100,23
do k=1,100,23
do i=1,100
do iT6_1=0,min(100-j,22)
tmp=a(i,j+iT6_1)
do iT6_2=0,min(100-k,22)
tmp=tmp+b(i,k+iT6_2)
      *c(k+iT6_2,j+iT6_1)
enddo
a(i,j+iT6_1)=tmp
enddo
enddo
enddo
enddo
```

(b)

```
do j=1,100,23
do k=1,100,23
do iw21_0=0,min(100-j,22)
do iw21_1=0,min(100-k,22)
c52(iw21_0+23*iw21_1)=
        c(k+iw21_1,j+iw21_0)
enddo
enddo
do i=1,100
do iw21_0=0,min(100-k,22)
b47(iw21_0)=b(i,k+iw21_0)
enddo
do iT6_1=0,min(100-j,22)
a37=a(i,j+iT6_1)
doiT6_2=0,min(100-k,22)
a37=a37+b47(iT6_2)*
    c52(iT6_1+23*iT6_2)
enddo
a(i,j+iT6_1)=a37
enddo
enddo
enddo
enddo
```

(c)

Figure 4: (a) Original matrix-matrix multiplication, (b) blocked matrix-matrix multiplication, (c) blocked matrix-matrix multiplication with copies

matrix-multiply: 5 reuses per data on arrays vx0, vy0, 3 reuses per data in array po[4] , and only one access to each element of arrays ivx, ivy, vxn and vyn; ivx and ivy are integer arrays. As there are 4 floating point words or 8 integer words per cache block, the first references misses on this seven arrays represent $\frac{5*340*340}{4} + \frac{2*340*340}{8} = 173400$ misses.

In terms of access to arrays, the overcost of copying is huge here. The three arrays vx0,vy0 and po have to be copied generating 905938 extra references on floating data: more than one third of the floating-point data references are done while copying! Paradoxically, the total number of memory references in the Blocked/Copy version is lower than in the other versions: the F77 compiler generates 12 references to scalars used for address generation in the original algorithm and the blocked algorithm. This is shown in table 2.

The numbers of misses for the leading size of the arrays ranging from 340 to 600 are given in Figure 7.

Original loop: The $10 * 340$ floating-point distinct elements and $2 * 340$ distinct integer elements are used in one iteration of the outermost loop, this exceeds the size of the cache. Then most of the data used in iteration j will be invalidated in the cache before it can be reused during iteration j+1. This effect can be seen on Figure 7.

[4] 4 reuses are present, but analysis of the assembler code confirms that one of this reuse is captured by register

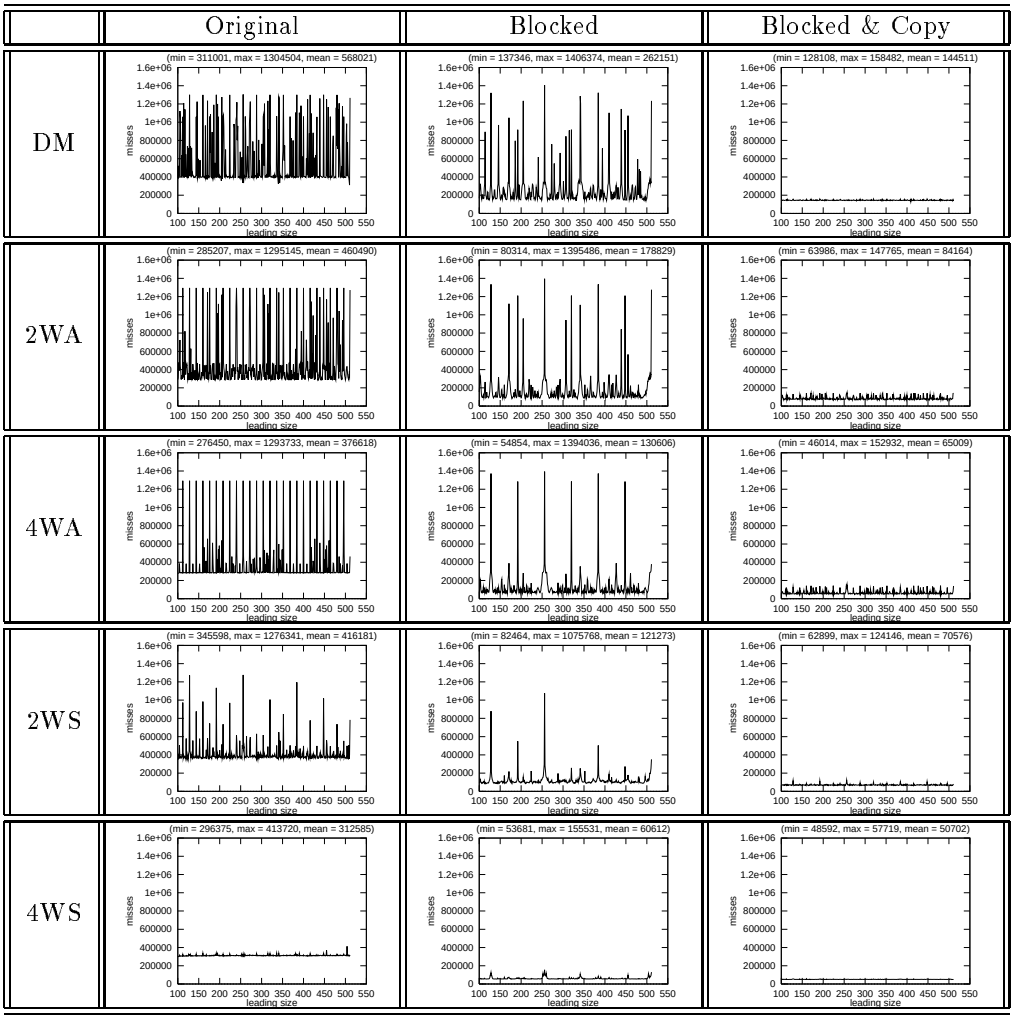


Figure 5: Matrix-matrix multiply. DM stands for Direct Mapped, 2WA for 2 way associative cache, 4WA for 4 way associative cache, 2WS for 2 way skewed cache, 4WS for 4 way skewed cache.

```
do  j = 1,340
   do  i = 1,340
temp = c0 * vxo(i,j) + dty2 * (vxo(i -1,j) + vxo(i + 1,j))
  + dtx2 * (vxo(i,j + 1) + vxo(i,j -1)) -dtx * (po(i,j) -
(po(i,j-1)) -c1
temp = temp * ivx(i,j)
vxn(i,j) = temp
temp = c0 * vyo(i,j) + dty2 * (vyo(i -1,j) + vyo(i + 1,j))
  + dtx2 * (vyo(i,j + 1) + vyo(i,j -1)) -dty * (po(i -1,j) -
   po(i,j)) -c2
temp = temp * ivy(i,j)
vyn(i,j) = temp
   enddo
enddo
```

Figure 6: 2D Jacobi loop nest

	Original	Blocked	Bl & Co
floating point ref	1734008	1734008	2639946
data ref	3520493	3837505	3418390
instructions	9295221	10066791	10789758

Table 2: Characteristics on the different 2D Jacobi versions

Blocked loop: As, for the matrix multiply, some pathological behaviors can be observed for usual cache structures, while the behavior of the 4-way skewed-associative cache is quite uniform.

Blocked and Copy: Copying arrays may also be considered to average the number of misses, but due to the high cost of copying and the small amount of temporal locality, this version of the loop exhibits high miss ratio : the average number of misses is 50% higher than in the original loop.

5.3 LU factorization

The last loop nest we experimented is a 100×100 LU factorization without pivoting. The three versions of the loop nest are shown Figure 8. The blocked version is due to S. Carr [3]. The blocked copied version was handly derived.

The blocked version of the kernel exploits temporal locality by reducing the working set to a set of contiguous columns of the matrix. So temporal locality can be exploited only if the set of columns can fit in the cache and are mapped in different places. The relative placement of the columns used in a block depends on the value of the leading dimension of the array `a`.

The characteristics of the three codes are given in Table 3. Copying adds 137196×2 floating point references, an increase of 20 % in the number of load/store operations.

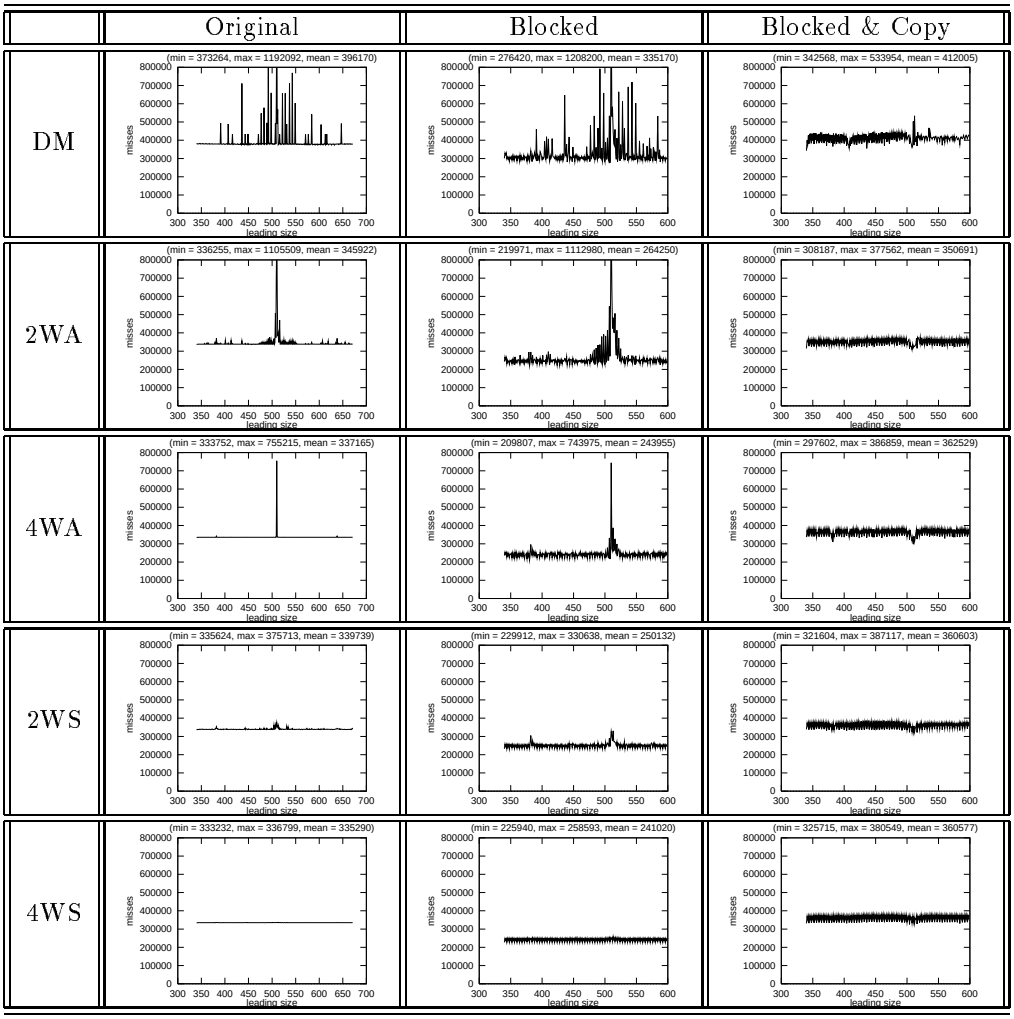


Figure 7: Simulation results for the 2D Jacobi loop

Moreover the increase in the number of instructions over the blocked version is around 48%; unless the performance predictability is the major issue for the user, such a copying is clearly unacceptable from the performance point of view.

The number of misses for the three codes is given in Figure 9.

Original loop: Only the direct mapped cache exhibits erratic behavior. In average the number of misses for the direct mapped cache is 15 % higher than on the other caches.

Blocked loop: Blocking is effective on this kernel but shown very high miss ratio for some value of the leading size for the direct mapped and set associative cache.

Blocked plus Copy loop: Copying is effective in reducing peak miss rate, however it increases the average number of misses except for the 4 way set-associative cache.

As with the previous experiments, 4-way skewed associative cache exhibits the best average behavior.

	Original	Blocked	Bl & Co
floating point ref	1328352	1328352	1602744
data ref	1488028	1488028	1763389
instructions	4123466	5439516	8103428

Table 3: Characteristics on the different LU versions

6 Use of cache space

In the previous experiments, the block size was computed so the data reused in a block fits in approximately half of the cache size. It is clear that the best blocking factor depends on the cache organization. The experiment presented in this section measures the influence of the blocking factor on the number of misses for a 96×96 blocked matrix-matrix multiplication.

The size 96×96 was chosen as an example, because the respective numbers of block matrix multiplies vary gracefully when the block size varies from 4 to 32. Statistics on the different blocked version executions are reported in Table 4. This table clearly shows that, when the block size increases, the overhead due to the loop blocking decreases in terms of numbers of floating point data references, integer data references and instructions; thus using a large block size is desirable, if it does not increase too dramatically the miss numbers.

The size of the data, subject to reused in a block, for the arrays A, B and C are respectively 1, bl, bl^2 with bl being the block size. In Table 5, we illustrate the required fraction of the cache for each block size used and the estimated number of misses for a 96×96 matrix-matrix multiplication which would be obtained assuming a perfect cache but no reuse across the blocks[5].

[5] This hypothesis is not realistic for very small block size

```
do k=1,n-1
do i=k+1,n
a(i,k)=a(i,k)/a(k,k)
enddo
do j=k+1,n
do i=k+1,n
a(i,j)=a(i,j)-a(i,k)*a(k,j)
enddo
enddo
enddo
```

(a)

```
do k=1,n-1,ks
do kk=k,min(k+ks-1,n-1)
do i=kk+1,n
a(i,kk)=a(i,kk)/a(kk,kk)
enddo
do j=kk+1,k+ks-1
do i=kk+1,n
a(i,j)=a(i,j)-a(i,kk)*a(kk,j)
enddo
enddo
enddo
do j=k+ks,n
do i=k+1,n
do kk=k,min(min(k+ks-1,n-1),
            i-1)
a(i,j)=a(i,j)-a(i,kk)*a(kk,j)
enddo
enddo
enddo
enddo
```

(b)

```
ks=5
do k=1,n-1,ks
do kk=k,min(k+ks-1,n-1)
do i=kk+1,n
a(i,kk)=a(i,kk)/a(kk,kk)
enddo
do j=kk+1,k+ks-1
do i=kk+1,n
a(i,j)=a(i,j)-a(i,kk)*a(kk,j)
enddo
enddo
enddo
do l=k,min(k+ks-1,n-1)
do m=k+1,n
buffer(m,l-k)=a(m,l)
enddo
enddo
do j=k+ks,n
do m=k,n
buffer2(m)=a(m,j)
enddo
do i=k+1,n
do kk=k,min(min(k+ks-1,n-1),
            i-1)
buffer2(i)=buffer2(i)
-buffer(i,kk-k)*buffer2(kk)
enddo
enddo
do m=k+1,n
a(m,j)=buffer2(m)
enddo
enddo
enddo
```

(c)

Figure 8: (a) Original LU, (b) blocked LU, (c) blocked LU with copy

block size	4	8	12	16	20	24	28	32
array ref.	2211840	1990656	1916928	1880064	1861632	1843200	1843200	1824768
data ref.	2538312	2192244	2095384	2050422	2028808	2007772	2007772	1987314
instructions	12221746	8622886	7546978	7032220	6780640	6532926	6532926	6289078

Table 4: Statistics on 96*96 blocked matrix multiplications with various block sizes

block size	4	8	12	16	20	24	28	32
cache fraction	2 %	7 %	15 %	26 %	41 %	58 %	79 %	103 %
misses	112896	57600	39168	29952	25344	20736	20736	16128

Table 5: Estimated minimum numbers of misses and fractions of the cache for various block sizes

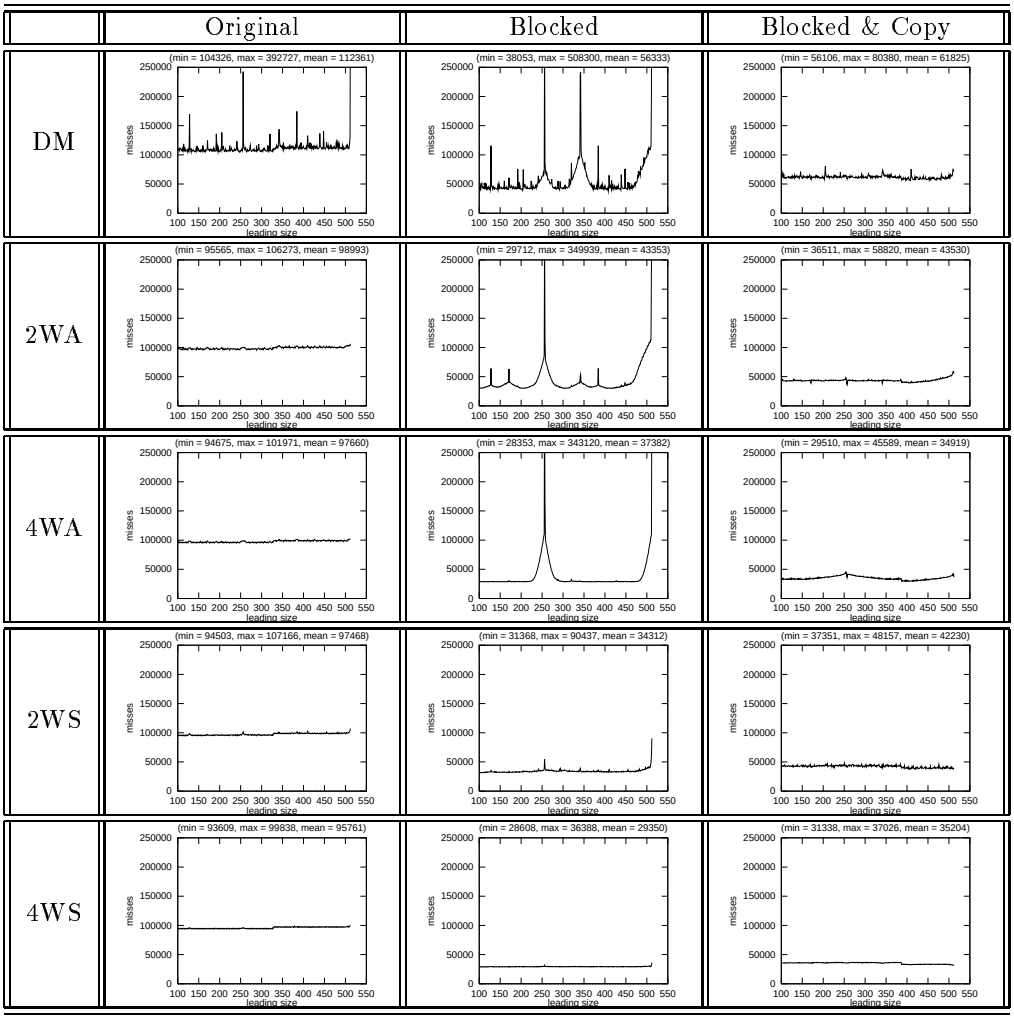


Figure 9: Simulation results for LU

Figure 10 illustrates the cache behaviors which were measured. The block size varied from 4 to 32 with a step of 4 and the leading dimension of the matrix was set from 200 to 300.

It can be seen on the curves that the block size has few influence on repartition of the peak miss numbers; the array sizes for which interferences appear are quite independent of the block size, but the magnitude of interference phenomena increases with the block size. As in previous experiments, it can be noted that the behavior of the 4 way skewed-associative cache is very regular and only exhibits some small irregularities when the required space for storing the reused blocks exceeds the cache size.

On Figure 10, we also illustrate the minimum and average number of misses in function of the block sizes.

Minimum miss numbers When using the direct mapped cache, the minimum miss number is obtained with a block size of 8; for the other caches, the minimum number of misses is obtained for a block size between 16 and 20. Notice that the 4 way set-associative cache exhibits a slightly lower minimum miss number (39547) than the 4-way skewed-associative cache (40351).

Average miss numbers For set-associative caches and direct-mapped caches, the shapes of the curves are not identical for the minimum miss numbers and the average miss numbers. For instance, for obtaining the lowest average miss number, a block size of 4 has to be used instead of a block size of 8 for the lowest minimum miss number. Notice also the large difference existing between the average miss numbers and the minimum miss numbers, e.g. for the 4-way set-associative cache a factor of two is observed for block size 20 (90291 versus 39547). For the 4-way skewed-associative cache, there is no such phenomenon; the mean miss numbers are very close to the minimum miss numbers for all block sizes.

Performance analysis As the overhead of loop blocking decreases when the block size increases, the choice of the optimum block size depends on the number of misses, but also on other parameters such as the instruction issue rate, the number of memory accesses and the miss penalty.

We use formula 1 to roughly modelize the number of cycles of an execution on an hypothetic superscalar processor issuing an average three instructions per cycle and paying a 10-cycle penalty on each miss

$$\frac{N_{inst}}{3} + 10 * N_{miss} \tag{1}$$

N_{inst} being the number of issued instructions and N_{miss} the number of misses.

Figure 11 illustrates the minimum and average numbers of cycles needed for executing the blocked matrix algorithm for different block sizes and cache organizations. Notice that for all cache organizations, except the direct-mapped, the minimum number is in the range

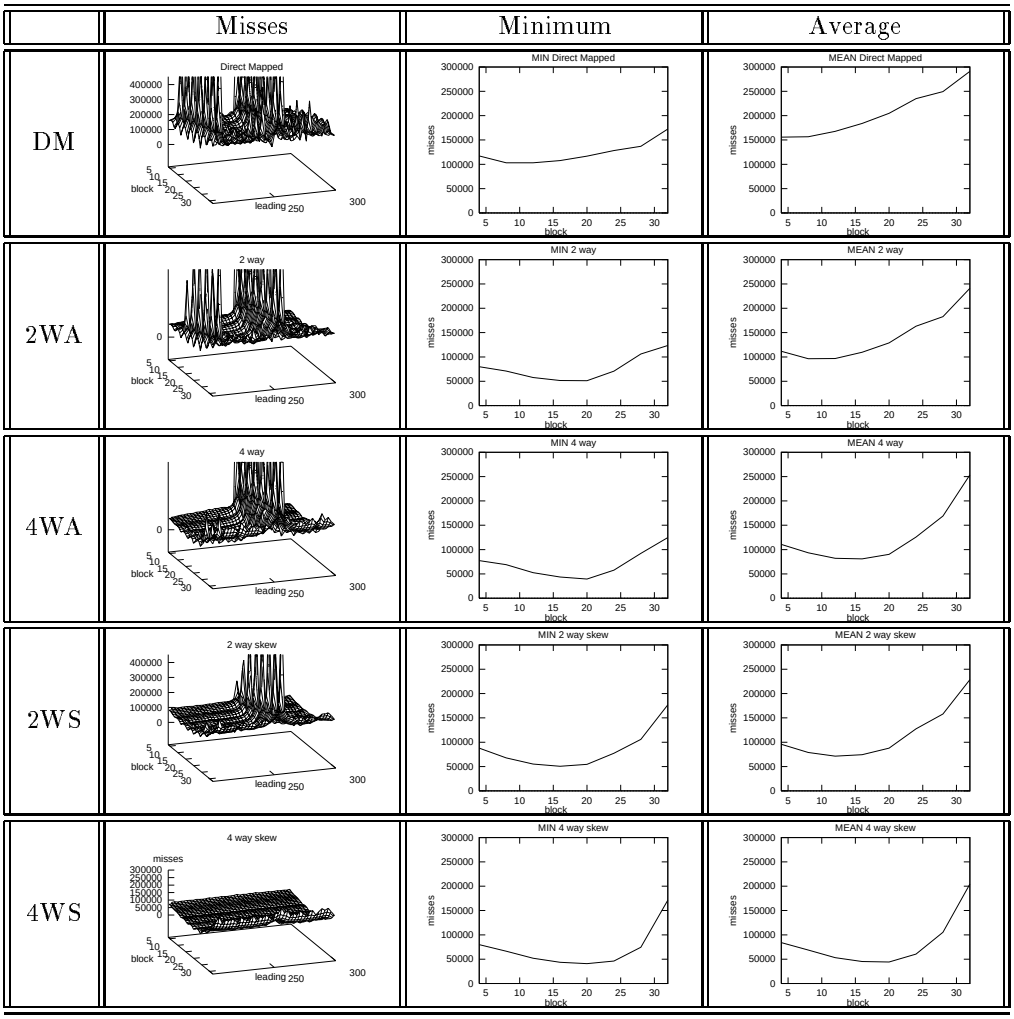


Figure 10: Simulations for Matrix multiplication with various block sizes (4 to 32) and different leading sizes (200 to 300)

of 2640000 to 2800000; but the minimum average number is significantly higher for 2-way and 4-way set-associative caches (resp. 3440373 and 3152793 cycles) while it is very close to the minimum for the 4-way skewed-associative cache.

Notice that, on set-associative and direct-mapped caches, for some array sizes, the execution times would be in the 15000000 cycles range, i.e. about 5 times the average execution time; such a phenomenon does not appear with a 4-way skewed-associative cache.

It must also be noticed that a blocking factor of 20 or 24 leads approximately to the same range of performance for the 4-way skewed-associative cache, then about 58 % of the cache space may be used for blocking instead of approximately 40 % of the cache size for the 4-way set-associative cache (blocking factor of 16 or 20).

7 Conclusion

Effective performance on today computers depends on many parameters. Performance tuning has become very complex. One of the factor of this complexity is the use of memory hierarchies, and particularly of caches. Code transformations such as loop blocking may be used for improving temporal locality in numerical codes; nevertheless, even when loops are blocked in a numerical application working on regular arrays, the behavior of direct-mapped and set-associative caches is very sensitive to parameters such as the sizes of the arrays. This leads sometimes to unpredictable and catastrophic performance. In the case of the direct mapped cache as shown in [10, 18, 19], cache interferences can sometimes be computed, allowing to use copying only when it is useful; unfortunately in many cases this is not possible (for instance, in numerical libraries). Computing cache interferences for set-associative caches is very complex, then very conservative criteria must be applied. Unfortunately, except when temporal locality is very high, the copying represents a significant overhead. However it is probably more efficient to change array declaration rather than to copy.

The high variation of the number of misses in the case of usual direct mapped caches and set-associative caches shows that for many numerical applications these are not adequate structures for such a code and makes any performance number very suspect as a representation of the performance of the processor. It should be interesting for the user of such caches to be aware of this phenomenon when performance is critical. At least for some kernel in library the user should have the set of values to avoid according to its particular cache configuration: it is noticeable that avoiding power of two as dimensions of arrays is not sufficient as recalled by the example in [12].

Most users are not expert in cache organizations and cannot be aware of such phenomena. In this paper, we have shown that the skewed associative cache recently proposed in [13, 14] may provide to the user a quite predictable cache behavior; moreover its average behavior is also better than the corresponding set-associative cache behavior. Our experiments have shown that the characteristics of the 4 way skewed-associative cache are such that it is quite insensitive on variations of the array placements in memory. Copying is never necessary to improve performance, when direct-mapped and set-associative caches all necessitate copying to improve cache behavior and get predictable performance.

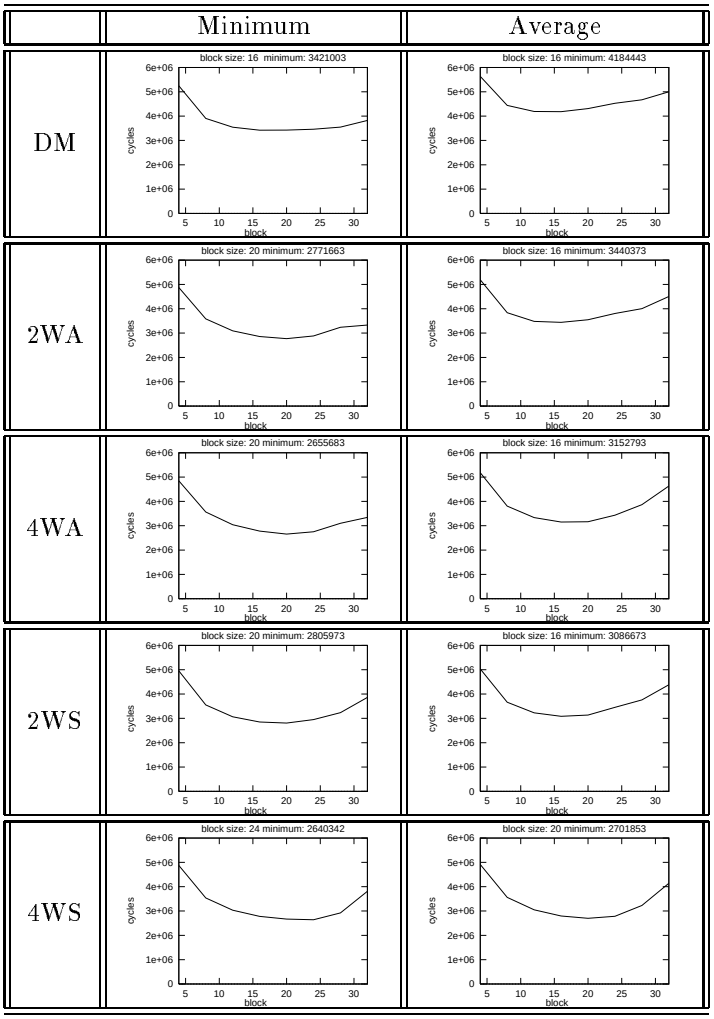


Figure 11: Estimated execution times for 96x96 blocked matrix multiply for various block sizes

Moreover as shown in Section 6, for blocked algorithms, the skewed-associative allows a better usage of the cache space than the set-associative cache. Then it allows to use a larger fraction of the cache space for blocking the loop, thus leading to the use of larger blocking factor and a reduced overhead due to blocking.

Our experiments have also emphasized the bad behavior of direct-mapped caches; even on blocked copi ed versions of the algorithms, the average numbers of misses on a direct-mapped are significantly higher than on a set associative cache and *a fortiori* on a skewed-associative cache.

References

[1] F. Bodin, C. Eisenbeis, W. Jalby, and D. Windheiser. A quantitative algorithm for data locality optimization. In *Code Generation-Concepts, Tools, Techniques*, Springer Verlag, 1992.

[2] Bernard D., Bodin F., Goasguen A., Fechant C., "Implementing a two dimensional pore-scale flow model on different parallel machines", To appear in Proceedings of X international Conference on Computational Methods in Water Resources, 19-22 June, 1994.

[3] Carr S. *Memory-Hierarchy Management*, PhD thesis, Rice University, February, 1993.

[4] Callahan D, Carr S, Kennedy K. *Improving Register Allocation for Subscripted Variables*, Proceedings of the Conference on Programming Language Design and Implementation, 1990.

[5] Eisenbeis C, Jalby W, Windheiser D, Bodin F. *A Strategy for Array Management in Local Memory*, Special Issue of Math. Programming B on Applications of Discrete Optimization in Computer Science, 1993.

[6] J.L. Hennessy, D.A. Patterson *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers, Inc. 1990

[7] M.D. Hill, "A case for direct-mapped caches", IEEE Computer, Dec 1988

[8] M.D.Hill, A.J. Smith "Evaluating Associativity in CPU Caches" IEEE Transactions on Computers, Dec. 1989

[9] G.Irlam "Spa" personal communication 1992; the Spa package is available from gordoni@cs.adelaide.edu.au

[10] Lam M, Rothberg E, Wolf M. *The Cache Performance and Optimizations of Blocked Algorithms*, Proceedings of the Fourth ACM ASPLOS conference, April 91, pp 63-75.

[11] Porterfield A. *Compiler management of program locality*, Technical Report, Rice University, Houston, Texas, January 1988.

[12] M. Schlansker, R. Shaw, A. Sivaramakrishnan "Randomization and Associativity in the Design of Placement-Insensitive Caches" HP Laboratories Technical Report 93-41, June 1993

[13] A. Seznec, "A case for two-way skewed associative caches", Proceedings of the 20th International Symposium on Computer Architecture, May 1993

[14] A. Seznec, F. Bodin, "Skewed-associative caches", Proceedings of PARLE' 93, Munich, june 1993

[15] A.J. Smith "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory" IEEE Transactions on Sofware Engineering, March 1978

[16] A.J. Smith "Cache memories" ACM Computing Surveys, Sept. 1982

[17] R.A. Sugumar, S.G. Abraham "Efficient Simulation of caches under optimal replacement with applications to miss characterization", In Proceedings ACM SIGMETRICS conference, 1993.

[18] Temam O., Fricker C. and Jalby W. *Evaluating the impact of cache interference on numerical codes*, ICPP, 1993.

[19] O. Temam, E. Granston, W. Jalby "To Copy or Not to Copy : A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts" Proceedings of Supercomputing'93 (ACM), Portland, nov. 1993

[20] Wolf M, Lam M. *An algorithm to generate sequential and parallel code with improved data locality*, Technical Report, Stanford University 1990.

[21] Wolf M, Lam M. *A Data Locality Optimizing Algorithm* ACM Conference on Programming Language Design and Implementation, June 26-28, 1991.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399